Holistic Debugging of MPI Derived Datatypes

J. Protze, T. Hilbrich, A. Knupfer, B. R. de Supinski, M. S. Mueller

January 4, 2012

International Parallel and Distributed Processing Symposium (IPDPS)
Shanghai, China
May 21, 2012 through May 25, 2012

Holistic Debugging of MPI Derived Datatypes

Joachim Protze[*], Tobias Hilbrich[*], Andreas Knüpfer[*], Bronis R. de Supinski[†] and Matthias S. Müller[*]
[*]*Center for Information Services and High Performance Computing*
TU Dresden, Dresden, Germany
{joachim.protze|tobias.hilbrich|andreas.knuepfer|matthias.mueller}@tu-dresden.de
[†]*Center for Applied Scientific Computing*
LLNL, Livermore, USA
bronis@llnl.gov

***Abstract*—The Message Passing Interface (MPI) specifies an API that allows programmers to create efficient and scalable parallel applications. The standard defines multiple constraints for each function parameter. For performance reasons, no MPI implementation checks all of these constraints at runtime. Derived datatypes are an important concept of MPI and allow users to describe an application's data structures for efficient and convenient communication. Using existing infrastructure we present scalable algorithms to detect usage errors of basic and derived MPI datatypes. We detect errors that include constraints for construction and usage of derived datatypes, matching their type signatures in communication, and detecting erroneous overlaps of communication buffers.**

We implement these checks in the MUST runtime error detection framework. We provide a novel representation of error locations to highlight usage errors. Further, approaches to buffer overlap checking can cause unacceptable overheads for non-contiguous datatypes. We present an algorithm that uses patterns in derived MPI datatypes to avoid these overheads without losing precision. Application results for the benchmark suites SPEC MPI2007 and NAS Parallel Benchmarks for up to 2048 cores show that our approach applies to a broad range of applications and that our extended overlap check improves performance by two orders of magnitude. Finally, we augment our runtime error detection component with a debugger extension to support in-depth analysis of the errors that we find as well as semantic errors. This extension to gdb provides information about MPI datatype handles and enables gdb – and other debuggers based on gdb – to display the content of a buffer as used in MPI communications.

***Keywords*-MPI; datatypes; runtime error detection; debugging**

I. Introduction

The Message Passing Interface (MPI) [1] is designed to work with heterogeneous environments and to support one-copy semantics. Therefore each communication call must specify datatypes that describe the data in its associated buffers. MPI's predefined datatypes (or *basic types*) represent the types that are available in common programming languages, e.g., `MPI_INT` for an integer in C and `MPI_INTEGER8` for an 8 byte integer in Fortran. The standard also provides derived datatypes that support transfers of more complex data regions, including non-contiguous patterns of mixed type data.

Table I
Example for a type mismatch

Task 0	Task 1
Type = (`MPI_INT`, 0) (`MPI_INT`, 4) (`MPI_DOUBLE`, 8)	Type = (`MPI_DOUBLE`, 0) (`MPI_INT`, 8) (`MPI_INT`, 12)
Send (to:1, type:Type, count:1)	Recv (from:0, type:Type, count:1)

MPI type constructors, which create derived datatypes, can describe all memory usage patterns with any random access order. Communcations that use derived datatypes require the MPI library to collect the data for transmission, which can facilitate the use of advanced hardware features such as DMA engines that perform scatter/gather operations. While MPI type constructors provide a powerful and flexible mechanism for describing data, their use is quite complex and error-prone. Detecting and locating erroneous MPI datatype usage can be very challenging. We have even found such errors in the well-known and widely used SPEC MPI2007 benchmark suite [2].

Table I sketches a usage error of derived datatypes that results in a datatype mismatch in point-to-point communication. Each task uses a distinct derived datatype that we illustrate with its type map, which is a set of tuples in which each tuple contains a basic type and a byte displacement relative to the communication buffer. For example, (`MPI_INT`, 4) refers to a C integer that starts at the fifth byte of the communication buffer. The MPI standard specifies that the basic datatypes of point-to-point communications must be pairwise equal, except that the receive buffer may contain additional unused basic types. Our example violates this condition as it tries to match the type `MPI_INT` to the type `MPI_DOUBLE`, which are clearly not equal. However, the datatype on rank 0 is 16 bytes in size and fits into the 16 bytes that the receive type spans. As MPI implementations are optimized for speed; they do not check for erroneous usage. Thus, implementations do not detect the erroneous communication, and instead just write the two integer values into the first double value.

We can detect the usage error in the previous example with automatic tools. However, semantic errors that adhere to the rules of the MPI standard but do not implement the intended behavior are more difficult to detect. Consider the faulty creation of a derived datatype that should represent a C-struct. Developers often subtract addresses of individual fields of the struct during the creation of the derived datatype. If the operands of the subtraction are switched by accident, then an example type looks like {(`MPI_LB`, 0), (`MPI_INT`, 0), (`MPI_DOUBLE`, -8), (`MPI_CHAR`, -16), (`MPI_UB`, -24)} instead of {(`MPI_LB`, 0), (`MPI_INT`, 0), (`MPI_DOUBLE`, 8), (`MPI_CHAR`, 16), (`MPI_UB`, 24)}. Although this datatype conforms to the MPI standard, it is semantically incorrect and will lead to sending or receiving invalid data when used in communication operations.

Parallel debuggers and runtime error detection tools exist that detect syntactic and semantic errors of MPI datatype usage. However, all currently existing runtime tools only support some correctness checks, have limited scalability, or may not detect some errors. Further, debuggers provide no details for MPI datatype handles, which significantly complicates datatype debugging. We present an approach to holistic MPI datatype debugging that uses runtime error detection and classical debuggers jointly to combine wide coverage with scalability. Our contributions include:

- A classification of MPI usage errors that can be detected at runtime;
- A novel visualization for derived datatype errors;
- Two approaches to MPI type matching to support efficient correctness checks for complex derived datatypes;
- The strided block concept to check for buffer overlap;
- Algorithms that support efficient overlap checking for heavily non-contiguous datatypes;
- A debugger extension to provide insight into datatype handles and to print communication buffer contents;
- Experiments that demonstrate the applicability and scalability of our approach.

Our experiments with two widely used benchmark suites show that our extended overlap check improves performance by up to two orders of magnitude. Further, they demonstrate that our tool scales to 2048 MPI processes, well beyond that of any existing MPI datatype error detection tools.

The remainder of the paper is structured as follows. Section II provides more background on MPI datatype usage errors, while Section III covers related work. Section IV presents different representations of MPI datatypes. We then detail our overall tool for detecting MPI usage errors in Section V. Section VI presents our algorithms to detect datatype mismatches in communication calls. Section VII covers our basic overlap check, while Section VIII presents strided blocks and the extended overlap check. Section IX evaluates our techniques in terms of performance and broad applicability and Section X presents our extension to gdb.

II. Usage Errors

The MPI standard provides type constructors. The simplest constructor `MPI_Type_contiguous` concatenates a given type while `MPI_Type_struct` is the most general constructor. Figure 1 illustrates the later constructor with an example. The individual constructors combine either one or multiple existing datatypes into a new one by using repetitions and displacements. MPI returns the newly created datatype as a handle. The attributes of the datatype are managed within the MPI library while the application uses the handle to reference the datatype. Type constructors can use derived datatypes and the new type remains valid even if one of its base types is freed.

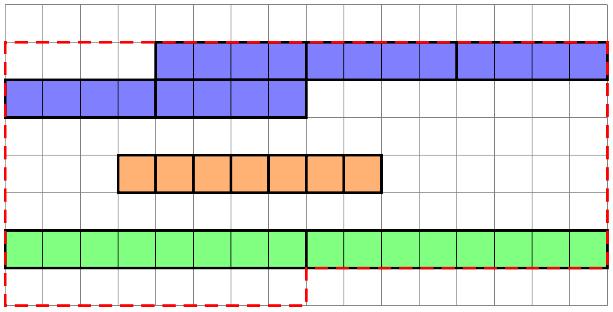


Figure 1. Example for a struct type: `B = {1, 5, 7, 2, 1};` `D = {-4, 0, 47, 76, 100};` `T = {MPI_LB, MPI_INT, MPI_CHAR, MPI_DOUBLE, MPI_UB};` `MPI_Type_struct (5, B, D, T, *newtype);`

The MPI standard defines several constraints for the use of these constructors and of any datatype:

- Argument restrictions for invoking type constructors;
- A derived datatype must be committed before being used in any communication calls;
- Types matching rules;
- Concurrent operations must not access overlapping memory locations if at least one has receive semantics;

Automatic correctness checks can easily detect violations to the first two constraints while violations to the latter two are far harder to detect.

The standard defines three transfer steps to which type matching rules apply. The first step transfers data from application memory to the MPI library. With few exceptions (i.e., `MPI_BYTE` and `MPI_PACKED`) the programming language base types of the locations must correspond to the MPI basic types. The second step transfers data to the communication partner. The sequence of basic types associated with the transferred data must be the same on both sides, although the receiver can specify a longer sequence than is sent. The third step transfers data from the MPI library to the application for which the MPI basic types and programming language base types must again correspond.

A communication operation that has receive semantics (e.g., `MPI_Recv` or `MPI_Alltoall`) implies a write to the associated buffer. Concurrent communications that access overlapping memory regions would imply a race condition if at least one has receive semantics, so they are erroneous. Older MPI versions imposed a stricter condition that allowed implementations to write buffer locations even for operations with send semantics. Non-blocking communication calls, which can use the communication buffer after the initial communication call, and collective calls, which combine send and receive semantics, complicate detecting violations of this restriction. Like any race condition, its violations do often not manifest despite repeated use of the code until they finally lead to a mysterious crash.

III. Related work

Several tools for debugging or error detection in MPI applications exist. Approaches for runtime error detection include ISP [3], Marmot [4], and Umpire [5]. ISP offers type matching for point-to-point calls while no publication details the algorithm in use. This approach performs no type matching for MPI collectives and does not detect invalid buffer accesses. Marmot offers buffer overlap detection for contiguous datatypes that can catch some invalid buffer accesses but does not check type matching. Umpire implements type matching for point-to-point messages and for collectives although its algorithms have not been published previously. Umpire also uses a checksum approach to detect invalid access to buffers. The overhead of this approach increases with buffer size and its precision is limited.

Another approach to type matching [6] generates a hash for type signatures. The MPI standard allows the specification of partial receives that only use parts of a given receive buffer. The hashing approach must adapt the calculation of the hash accordingly, which can cause significant increase of overhead. Also, hashes must be piggybacked with actual messages, possibly within the MPI library. Further, hash collisions can lead to false negatives. Finally, this approach cannot detect invalid buffer accesses.

MPI-CHECK [7] checks transfers between application memory and the MPI library. These checks require compiler analysis, which is limited to Fortran in MPI-CHECK. Further, the support appears to be limited to MPI basic types. In any event, this work is orthogonal to ours, which supports detection of all other MPI datatype errors.

Valgrind [8] provides powerful features for debugging memory-related bugs. Its MPI extension detects invalid accesses to application memory that is actively used in communication functions. This extension requires the MPI-2 functions `MPI_Type_get_envelope` and `MPI_Type_get_contents` but only supports MPI-1 derived datatypes. It handles derived datatypes with a recursive function. This function uses `MPI_Type_get_envelope` and `MPI_Type_get_contents` more often than required to determine the derived type structure. The functions are invoked for each repetition of an underlying type. Further, the Valgrind extension uses the recursive function for each communication call, which can cause considerable overhead for complex datatypes. However, the resulting correctness check is extremely precise although it does not check type matching and is unavailable for some architecture.

In summary, individual approaches to the detection of type mismatches and invalid buffer accesses exist. However, none of these approaches offers a convenient visualization of errors that involve complex datatypes. To the best of our knowledge, no single existing runtime tool both detects invalid accesses and type mismatches in a scalable way.

We need debugger support to investigate details of errors that automatic tools detect and to inspect semantic errors. All existing parallel debuggers (e.g., DDT [9] and Totalview [10]) treat MPI handles as opaque handles. The debuggers only display the numeric values of the handles and do not provide any mechanism to examine the actual MPI object. No existing debugger supports MPI datatypes, not even the basic types, which have constant handle values.

The MPI forum is designing a tools information interface for MPI-3 to provide tools access to information on MPI handles. Enhanced features of debuggers could use this interface to highlight the values that will be communicated when using a given datatype and buffer. However, the new interface is not yet available in any MPI implementation.

IV. MPI datatype display and element addressing

We can map datatypes built with MPI type constructors to trees [11]. These trees give a full or partial representation of the corresponding datatypes. We introduce several useful mappings that store the information used to construct the type. As we show in the final part of this section, we can use them to address an element of a datatype or buffer.

A. *Definition: meta type-tree*

The simplest mapping, a *meta type-tree*, projects the datatype to a tree in which nodes are datatype handles and the edges represent handle uses in the type's constructor. All leaves in this mapping represent basic types. This tree indicates the dependencies of the defined datatypes. The nodes can store the values for blocklength, displacements, and kind of type with the handles. Figure $2(a)$ shows the meta type-tree for the datatype that Figure 1 shows.

B. *Definition: expanded type-tree*

Another mapping, the *expanded type-tree*, projects the datatype to a tree that has more levels and many more nodes, than the meta type-tree. Each branching factor in the type constructor results in an equivalent number of edges starting from the node or the child node. For example, a vector type with the parameters *count*, *blocklength* and *oldtype* has *count* child nodes as an intermediate stage. Each intermediate stage has *blocklength* child nodes of type *oldtype*. The leaves of this tree represent basic types. For the example from Figure 1, we derive the expanded type-tree from its meta type-tree by inserting intermediate nodes where the counts are displayed and connect these to count copies of the attached leaves, as Figure $2(b)$ shows.

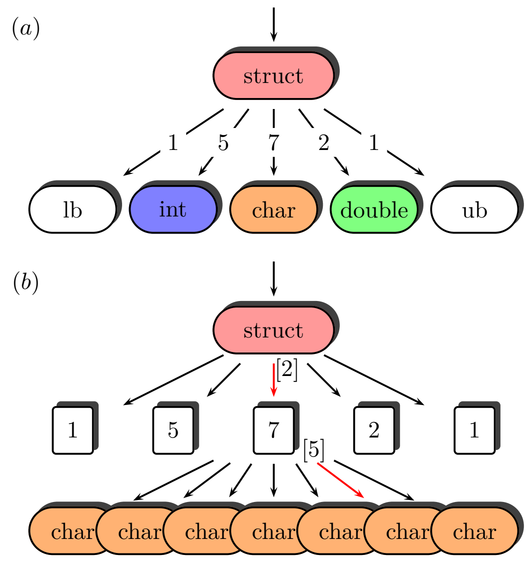


Figure 2. Examples of a meta (a) and expanded (b) type-tree

C. Addressing elements of datatypes

In various contexts, such as to point to the location of a violation, we must specify a single element of a derived datatype. Our method based on the expanded type-tree compactly addresses a precise element. Starting at a given datatype handle (i.e., a root node), we can determine the route through the tree by counting the edges to the left of node's incoming edge. For each step this count is given in square brackets and appended to the address. For the struct type *s_type* given in Figure 1, $s\_type(STRUCT)[3]$ points to the block of doubles while $s\_type(STRUCT)[2][5](CHAR)$ points to the sixth character, as Figure $2(b)$ shows. The datatype variable provides a reference point; the specification of the nodes in round brackets is only for clarification, so $s\_type(STRUCT)[2][5](CHAR)$ and $s\_type[2][5]$ are valid addresses of the same element, assuming that *s_type* is a valid handle of a datatype with proper sizes and structure.

In the long form, the intermediate stages manifest as succeeding brackets without kind specification in between. All MPI-1 datatypes other than contiguous have one intermediate stage. The contiguous type does not require an intermediate stage. MPI-2 datatypes need special consideration. The indexed-block type is handled as the indexed type. The resized type does not even require a node as it always has exactly one outgoing edge. For completeness, we print the resized type with one child [0]. For the darray and subarray datatypes, the dimension parameter *ndims* specifies the number of stages to the child type, so the number of intermediate nodes varies for these types.

V. Correctness Checking with MUST

We present an overview of the automatic correctness checks that we provide with the runtime error detection tool MUST, the Marmot Umpire Scalable Tool, in this section. MUST builds on the Generic Tool Infrastructure [12], which supports efficient tool development and application-specific tool instantiation. We target all four types of datatype usage errors that we discussed in Section II.

When we detect an error we provide the description of the problem, information on the MPI calls involved, and the datatype positions from Section IV (if applicable). Further, MUST provides as much background information as possible, e.g., if it detects an erroneous datatype usage, it lists information on the calls that created the datatype.

We first summarize our checks for argument restrictions and datatype state restrictions. We then introduce our message matching design. We describe our type matching algorithms in the next section, while Section VII describes our novel approach to detect buffer access violations.

A. Argument Restrictions

Marmot [4] provides various checks for argument restrictions and datatype commit state, which we adopt in MUST. These checksfall into the following classes:

1) Type constructor values: For arguments of type constructor invocations, we check the following conditions:

- Values of *count* or *blocklength* must be non-negative;
- Values of *ndims* in `MPI_Type_create_subarray` must be positive;
- Type constructor array parameters must be non-NULL.

2) Commit datatype handle values: A datatype handle that is used in an `MPI_Type_commit` invocation must not be `MPI_DATATYPE_NULL` or any handle that is not the result of a previous constructor call. The handle can have been committed previously or it can signify a basic type; we provide warnings for these unnecessary operations. Datatype handles used in `MPI_Type_free` must conform to the same restrictions although freeing a handle twice is an error and not just a warning.

3) Valid datatypes for communication: A datatype handle used in a communication operation must be committed and must not have been freed.

4) Datatype leaks: Derived datatypes should be freed before `MPI_Finalize` is invoked. MPI implementations use memory to track datatypes and other MPI objects so resource leaks are hidden memory leaks that can cause unexpected behavior.

B. Advanced Checks

While we could verify the above conditions on the application processes, some checks require additional communication. For example, type matching of the transfer to a communication partner must compare the send type with the receive type but each process only has information about one of the types. As previously discussed, Gropp [6] calculated a hash value from datatypes, which he prepended to each send call. He then checked in the receive operation that the hash value of the receive type matched.

ISP [3] and Umpire [5] use a different approach. These tools also perform runtime deadlock detection, which requires emulation of MPI message matching in order to determine which communication calls are active and which calls can complete. Piggybacking additional data with messages is infeasible since it would hang in the case of an application deadlock. Thus, both tools emulate message matching in a centralized manager. The manager detects any type mismatches when it detects a match.

We use the emulation approach since MUST includes deadlock detection. Thus, we run checks such as message matching within an MPI message matching emulator. A centralized message matching implementation can scale to about one thousand processes. To increase scalability, our other work is developing the first distributed implementation [12].

In the following sections we use the terms *type map* and *type signature* with the following meanings:

1) Definition: type map: The MPI standard defines a typemap as a sequence of pairs of an MPI basic type and a displacement. The displacements are in bytes and are either relative to the buffer address or are absolute, if the user specifies `MPI_BOTTOM` as a buffer. We can represent all derived datatypes as a typemap. All communication calls include a *count* argument that specifies how often to repeat the typemap. When we refer to the type map of a communication call, we mean the type map that results from repeating the type map of the given type *count* times.

2) Definition: type signature: A type signature is a sequence of MPI basic types that represents a datatype. It corresponds to a typemap but omits the displacements. When we refer to the type signature of the communication call, we mean type signature that results from repeating the signature of the given type *count* times.

VI. Type Matching

The MPI standard requires that the sender and receiver of communications use basic types with identical names (or `MPI_PACKED`) for each transmitted value. Correctness checks for this constraint must compare the information about the send type and the receive type to ensure that the type signatures of both communication calls are equal. For point-to-point calls, the type signature of the sending communication can be a proper prefix of the type signature of the receiving communication, in which case it is a *partial receive*, i.e., not all basic types of the type signature are used on the receiver side. Partial receives are illegal for collective communications. In the following, we first present MUST's default type matching algorithm. We then provide a second algorithm for more extensive use of derived datatypes.

A. Typesig Algorithm

Our basic algorithm uses a *typesig*, which has one entry for multiple entries of the same basic type.

1) Definition: typesig: A typesig, a compressed type signature, is a sequence of pairs that contain an MPI basic type and a repetition count. The count specifies how often the basic type is repeated in the type signature. We compare minimal size typesigs, which summarize all succeeding and equal basic types into one entry, i.e., no two succeeding entries have the same basic type. We compute the minimal sized typesig when we intercept an MPI type constructor.

2) Typesig-based type matching: Since each minimal typesig is compressed, we can compare two typesigs by comparing their entries pairwise for equality. As communication calls specify a *count* value to repeat a datatype, we must include its effect in the comparison. We could calculate new typesigs from the given types and counts and compare them directly. However, this approach may require additional storage and imposes unnecessary overhead. Thus, we loop count times over the typesig of the datatype. With this approach, the check is successful if we reach the end of both typesigs simultaneously since further comparison would yield no new insights. This typesig approach can check many derived datatypes in $\mathcal{O}(1)$. Most applications use derived datatypes that use only one basic type in their constructors, in which case the typesig only has one entry. This condition holds for all benchmarks in Section IX.

B. Regular Typesig Algorithm

If an application uses mixed basic types, e.g., to cover an array of C structs, then the typesig approach may become time consuming. We develop the concept of a regular expression representation of type signatures within Umpire [5]. We refer to these advanced type signatures as *regular typesigs*. Compared to the typesig approach, regular typesigs significantly reduce runtime overhead during type matching but can incur more overhead to prepare the representation when the datatype is committed.

Representation: The regular typesig is organized in a tree structure. The leaf nodes carry basic datatypes with a count for their repetition. The inner nodes can have any number of children and also carry an attribute for repetition.

Preparation: We use an unambiguous representation to simplify the comparison of regular typesigs. Algorithm 1 ensures that equal type signatures are represented by equal regular typesigs. For example, assume that d and c are basic types, a type signature with the pattern $dcdcdccc$ could be represented by $(dc)^3c^2$ or $(dc)^2dc^3$. Algorithm 1 generates the latter representation as it starts with the compressed pattern $dcdcdc^3$ and summarizes the first two pairs as a second step. For large alternating datatypes this algorithm may require a complete serialization in order to derive a regular typesig. An improvement would use this algorithm for the outermost struct. The other types just contribute a factor to the multiplicity of the root node.

Comparison: When we compare two regular typesigs, we strip the outermost multiplicity. The comparison of these

Algorithm 1 Preparation of a regular typesig.

```
$sigList \leftarrow$ compressedTypesig($type$)
$windowSize \leftarrow 2$
while $windowSize < \text{size}(sigList)/2$ do
    walk with two windowSized slides over sigList and summarize identical adjacent slides
    if summarized slides then
        $windowSize \leftarrow 2$
    else
        $windowSize \leftarrow windowSize + 1$
    end if
end while
```

Procedure 2 compare() - Compare regular typesigs.

```
in-parameter: $subtree1$, $subtree2$
out-parameter: $remain1$, $remain2$
if $subtree1$ and $subtree2$ are leaves then
    if $subtree1.type \neq subtree2.type$ then
        return mismatch
    end if
else if $subtree1$ or $subtree2$ is leaf then
    special handling needed
else
    for all $child1$, $child2$ from $subtree1$, $subtree2$ do
        compare ($child1$, $child2$, $remain1$, $remain2$)
        if $remain1$ or $remain2$ then
            special handling needed
            break
        end if
    end for
end if
if subtree1.count not subtree2.count then
    recalculate $remain1$ / $remain2$
end if
```

stripped typesigs provides the following results:

- Full match of the types;
- Mismatch;
- One fits the other (partially) with a given multiply.

An algebra determines the result of the actual type match query based on these return values. For the comparison of the stripped typesigs we can exploit the tree structure. Two typesigs are equal if all subtrees are equal, which we determine with Algorithm 2, that recursively walks the tree and compares the child nodes and their multiplicities. Full matches, which can be considered the general case, imply a minimal workload. Mismatches are also quickly determined. Partial matches require special handling. The remaining values carry the number of basic elements remaining for this subtree. If a remaining value arises and none of the current subtrees is finished, we also have a mismatch.

VII. Checks for overlaps

When an application issues a communication call, it passes one or more communication buffers associated with datatype(s) and repetition count(s) to MPI. If the call has receive semantics, the memory regions that are defined by these arguments must not be accessed until the communication completes. The following complexities make detecting violations of this restriction non-trivial:

- Non-blocking communications use memory buffer even after the initiating communication call returns;
- Datatypes may be self-overlapped;
- Collective calls can specify multiple buffers and have both send and receive semantics;

Our approach tracks memory regions that are used in all currently active communication calls. If a communication call passes a new memory region to MPI, we check whether it overlaps with a region that is already in use. If so we issue an error if at least one of the communications receives data. Thus, we detect all invalid accesses that result from specifying overlapping communication buffers.

We must track all memory regions that any active communication uses in order to detect invalid overlaps. Since derived datatypes can span non-contiguous memory regions, we may need to track a large number of memory intervals. However, we can detect these overlaps directly on the application processes as they involve only process-local information. We present a blocklist-based algorithm in this section and a further optimized algorithm in the next section. We start with additional terms and definitions, then present the detection of self-overlaps, and finally the detection of any overlap between any two active communications.

A. Definitions

1) Definition: buffer: As stated above, an MPI communication specifies the start address of the buffer, a datatype, and its repetition count. The start address is a position in memory relative to which the datatype defines a memory pattern. The count specifies the multiplicity of the transmitted datatype. The *buffer* is the composition of the initial buffer address, the datatype, and the count, i.e., the set of memory regions that this triple defines.

2) Definition: active buffer: A buffer is *active* if an active MPI communication is using it. The standard defines various communication types, including non-blocking point-to-point calls. A regular point-to-point call is active while the corresponding function call is active. A non-blocking communication is active until a completion call (e.g., `MPI_Wait`) marks the communication as completed. Collective communications require special handling since they collect multiple communications into one function.

3) Definition: overlap: Two buffers *overlap* if they contain at least one identical memory location. Also, datatypes may specify memory patterns that overlap, which we refer to

Algorithm 3 Create blocklist

```
childblocklist ← blocklist(childtype)
isSelfOverlapping ← isSelfOverlapping(childtype)
myblocklist ← ∅
for all childnode of datatype do
    for all interval ∈ childblocklist do
        insert (interval + offset of childnode) in myblocklist
    end for
end for
for all interval ∈ myblocklist do
    if interval.end ≥ interval.next.start then
        if interval.end > interval.next.start then
            isSelfOverlapping ← true
        end if
        concat(interval interval.next)
    end if
end for
```

as *self-overlapping* datatypes. Finally, if a buffer uses a self-overlapping datatype or a datatype that is self-overlapping at some repetition counts then the buffer is self-overlapping.

4) MPI standard: allowed and forbidden overlaps: The MPI standard permits overlaps in send buffers. Therefore self-overlapping datatypes may be built and committed. However, the use of self-overlapping buffers in receiving communications is erroneous, irrespective of whether the overlapping memory location is used by the receive.

5) Definition: blocklist: We can represent the memory regions of each buffer with a list of block intervals, called a *blocklist*. A blocklist is a set of pairs in which each pair defines a memory interval. The first value of each pair is the memory location that starts the interval and the second value is the first memory location that does not belong to the interval. For native datatypes, the blocklist consists of one interval that starts at 0 and ends at $0 + \mathit{length}(\mathit{BasicType})$. We can calculate blocklists for derived datatypes from the blocklists of their child type(s), the extent of the child type(s), and other values that specify the derived type, e.g., blocklengths and offsets. We assume that blocklists are sorted at all times, which allows more efficient overlap detection. We sort them by their first value. A blocklist is *compact* if it does not include any overlapping intervals. As an example we give the blocklist for the type displayed in Figure 1: {(0, 19), (47, 53), (76, 91)}

B. Blocklist Generation for Derived Datatypes

To improve the performance of overlap detection, we assume all blocklists are compact and sorted. If the blocklist is self-overlapping, i.e., the blocklist of a derived datatype that is only correctly used in send calls, we attach an extra *is-overlapping* flag to it. Thus, we can use a compact representation even for self-overlapping blocklists.

Algorithm 3 produces our compact blocklist. When we calculate compact blocklists for derived datatypes, we start with an empty blocklist and repeatedly add blocklists from the given base types to it. Each addition depends on the given type constructor. Most type constructors imply a time complexity of $\mathcal{O}(n)$ where $n = \mathit{count} \cdot \mathit{blocklength}$. The insertion of a single element is done in amortized constant time as we are merging sorted lists. The first part of Algorithm 3 shows the blocklist calculation for a derived datatype. The first *for* loop is dependent on the actual type constructor. For example, it loops over the given *count* and adds the blocklist of the base type in each iteration with an offset that is based on the extent of the base type for `MPI_Type_contiguous`. The algorithm then compresses the blocklist by combining subsequent intervals, which has time complexity $\mathcal{O}(n)$, as the blocklist is sorted. The algorithm also stores whether the new blocklist contains overlaps; if so it sets the overlapping flag to true.

C. Overlap Detection

The second part of Algorithm 3 illustrates how to detect overlaps between blocklists. In the general case with blocklists of length *n* and *m* this check has a time complexity of $\mathcal{O}(n+m)$. The property of compactness ensures the shortest possible serial representation of a blocklist, which means that the sizes of *n* and *m* are minimal.

1) Self-overlapping datatype check: Buffers of receiving communications must not use self-overlapping datatypes. We check this condition in constant time since the blocklist for the datatype already stores whether it is self-overlapping.

2) Self-overlapping buffer check: In order to check for a self-overlapping communication buffer, we must consider the repetition count. The datatype is repeated with an offset that is equal to the extent of the datatype. A precondition for an overlap is that the true extent of the datatype is larger than the extent; otherwise the repetitions are placed side by side. Further, the quotient $\lfloor \frac{\mathit{true\_extent}}{\mathit{extent}} \rfloor$ is the upper bound of repetitions in which overlaps can occur. The subsequent repetition is placed immediately next to the first. To test for an overlap for a given repetition, we repeat the blocklist for the datatype $\lfloor \frac{\mathit{true\_extent}}{\mathit{extent}} \rfloor$ times with a stride that is equal to the extent of the datatype. We then check the resulting intermediate blocklist for overlaps as in Algorithm 3.

We cache the result of this calculation in the context of the datatype. The results for a count times repetition can be used for other counts as well as the current one. If a count does not produce overlap, all smaller counts do not produce overlap. If a count produces overlap, all larger counts do also. Thus, we only cache two values for buffer self-overlap checks for each datatype: the largest count that has not produced overlap and the smallest count that has. For all checks that use any count that is between these two values, we perform the self-overlapping check and update the cached values, which we initialize to 0 and ∞.

3) Overlapping buffer check: The final overlap checks must cover all active buffers. While the previous overlap checks do not need to consider actual memory addresses, this check must consider the buffer address as well as the blocklist that results from the datatype and count. Thus, we use a memory-map that we derive from a compact blocklist by adding the buffer address to each entry. We also must store whether an entry in the memory-map is used with send or receive semantics, for which we use an extra flag. We must handle three groups of communications: 1) blocking communications, which must not overlap pending requests; 2) non-blocking communications, which must not overlap pending requests and must be added to pending requests; and 3) collective communications, which must not overlap pending requests and must not overlap with themselves. When MPI-3.0 adds non-blocking collective communications, we will need to add them to the set of pending requests in addition to the two overlapping checks. While checking these constraints, we must not report overlap of two send buffers as an error.

We merge the memory-maps of all pending requests into one set, that is sorted but not compact, which simplifies removing the entries that belong to a completed request. We can check any buffer for overlaps with pending requests by walking over the set of pending requests and the memory-map according to the buffer in question. This test has complexity $\mathcal{O}(n+m)$ where n and m are the sizes of the memory-map and the set of pending requests.

The handling of collective communications is specific to each call, so that we check the part of the buffer(s) that receiving processes use. We merge the memory-maps of the active buffers and check for overlaps as described above. Algorithm 5 contains an optimization – the inner while loop – to fast-forward over harmless blocks of pending requests. If the buffer is a send buffer, this while loop also fast-forwards over pending blocks marked for send.

Since we re-sort the blocks in the blocklist when we find an overlap, we cannot refer to an address within the datatype structure. We solve this problem by adding a further attribute to each blocklist entry that gives the byte position of the block when the buffer is serialized for transmission. This bijective projection of the datatype to integers is an appropriate representation within the internal data structures. Alternatively, we can easily translate this position to the address output format by recursing over the tree levels and using the child node's *size* value to locate the matching edge. In summary for each found overlap or typesig mismatch, we report the source code location and the exact address of the violating element within the datatype structure.

Algorithm 4 Check for overlap with pending requests

```
requestMemmap ← sortedListOfPendingBuffers
memmap ← memorymap(buffer)
for all block ∈ memmap do
    while leftmostBlock ∈ requestMemmap begins left of block's end do
        while leftmostBlock ∈ requestMemmap ends left of block's begin do
            remove leftmostBlock from requestMemmap
        end while
        if leftmostBlock ends right of block's begin and leftmostBlock begins left of block's end then
            found an overlap
        end if
        remove leftmostBlock from requestMemmap
    end while
end for
```

Algorithm 5 Check send buffer for overlap with pending requests

```
...
for all block ∈ memmap do
    while leftmostBlock ∈ requestMemmap begins left of block's end do
        while leftmostBlock ∈ requestMemmap is marked for send or ends left of block's begin do
            remove leftmostBlock from requestMemmap
        end while
        ...
    end while
end for
```

VIII. Strided blocks

When we tested our overlap checks we saw enormous overheads for *121.pop2* and *132.zeusmp2* (We introduce these benchmarks in detail in Section IX). For *132.zeusmp2*, we observed an overhead of 345% for the *lref* input set at 256 nodes. We identified comb-like datatypes as the source of this overhead. These datatypes describe planes of a multi-dimensional array. For a 3-dimensional array with edge length k, each plane contains $k \times k$ basic type entries. Depending on the memory layout of the plane, it can require as many as k^2 blocks in our blocklist representation. Thus, our overlap checks have time complexity $\mathcal{O}(k^2\, ln(k))$. We introduce the concept of strided blocks to reduce this cost.

A. Definition: strided-block

A *strided-block* replaces one or multiple blocklist entries. Where each blocklist entry defines a single interval, a strided-block defines a set of intervals that have a constant offset, i.e., a comb-like memory region. A strided-block is characterized by its interval and the length, count and stride of its blocks. The strided-blocklist is an ordered set of strided-blocks. In most cases we can decide with constant effort if a block or a strided-block overlaps with a strided-block by arithmetic evaluation.

Procedure 6 Check ***b***lock for overlap with ***s***trided-block

```
in-parameter: $b$ - block, $s$ - strided-block
if $s.end \leq b.begin$ and $s.begin \geq b.end$ then
    // block out of the region
    return false
else if $s.count = 1$ then
    // strided is compact
    return true
else if $b.begin < s.begin$ then
    // block begins left of stride
    return true
end if
// begin of block lays in i-th stride-interval
$i \leftarrow (b.begin - s.begin)/s.stride)$
if $b.begin < s.begin + i * s.stride + s.blocksize$ then
    // begin of block lays within the i-th stride-interval
    return true
else if $b.end > s.begin + (i + 1) * s.stride$ then
    // end of block lays in the (i+1)th stride-interval
    return true
end if
return false
```

B. Handling strided-blocks

Procedure 6 sketches the basic test for an overlap between a block and a strided block. We check two strided-blocks for overlap in constant time by considering these cases:

- The covered intervals do not overlap;
- The gap in strides is smaller than the blocklength:
  $\rightarrow$ The strided blocks collide on first intersection.
- The strides are similar in their factors:
  $\rightarrow$ The strided-blocks may lay side-by-side infinitely.

We handle these cases, which cover most use-cases, with constant effort. We only arrive at the remaining case if the two combs cover the same region in memory and have unusual stride values. This situation results in the worst case: we must test each block represented by the strided-block against the other strided-block, which has complexity $\mathcal{O}(min(n, m))$ where n and m are the count of blocks in the overlapping area of the two strided blocks.

C. Use of strided-blocks

Upon a closer look the concept of strided-blocks can exploit the potential of regularities in datatypes. Basically, the strided-blocks use the property of repetition in the construction of datatypes. For each kind of derived datatype this repetition manifests in terms of count, blocklength, subsizes or distribution. The only disadvantage of using a strided-block where we could use a single interval is the minimal memory overhead. Therefore we replace the concept of blocks generally by the concept of strided-blocks.

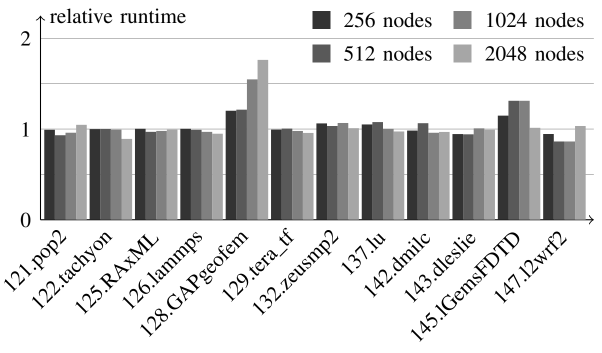


Figure 3. Overhead of MPI datatype checks to SPEC MPI2007

D. Further improvements

For some advanced MPI programs in production the use of strided blocks is not enough for satisfying runtime results. An example is a framework for dynamic load balancing that repeatedly exchanges irregular border areas to neighbor processes[13]. Despite using strided-blocks, the generated lists grow large. We could organize the strided-blocks in a binary search tree in which each node provides a bounding box for the intervals in this subtree to reduce the effort of checking for overlaps.

IX. Results of our automatic checks

We first present some errors that we detected in two SPEC MPI2007 [2] benchmarks. We then detail overhead measurements for our checks for overlaps for two benchmark suites: SPEC MPI2007 and the NAS Parallel Benchmarks [14].

A. Datatype errors in SPEC MPI2007

While measuring our tool's overhead, we detected bugs within the 115.fds4 and 122.tachyon. Both benchmarks use overlapping buffers in `MPI_Allgather` calls. For the case in which the send and receive buffers are equal, MPI-2 introduced the `MPI_IN_PLACE` argument. Both benchmarks intend this behavior but explicitly give the buffer address for both arguments, which is erroneous. 115.fds4 uses the same erroneous behavior for `MPI_Gather` calls.

B. Overhead

We measure our tool's overhead when applied to two standard benchmark suites: SPEC MPI2007 [2] and the NAS Parallel Benchmarks [14]. These benchmark suites come with various problem sizes. We used an 864 node Opteron Linux cluster with a DDR InfiniBand network for all experiments. Each node has 16 cores on four sockets and 32 GB of main memory that is shared between all cores. We used MVAPICH-1 for the measurements, although our tool is portable across MPI implementations. Since we aim to support large scale applications, we use the largest problem sizes available and with process counts of 256, 512, 1024

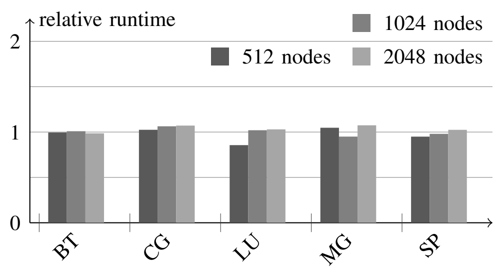


Figure 4. Overhead of MPI datatype checks to NPB

and 2048 nodes, which is lref for SPEC MPI2007 and class E for the NAS Parallel Benchmark suite. As the kernels *bt* and *sp* require square numbers as process counts, we use 529 and 2025 instead of 512 and 2048. Figures 3 and 4 show our measurement results. For each run, we give the relative runtime of the instrumented benchmark to an uninstrumented reference run. For many benchmarks the overhead is below 20%. For the kernel 128.GAPgeofem we trace the overhead to the massive use of collective operations – more than 1000 per second – and our tool performs the checks for each.

X. MPI DATATYPES WITHIN GDB

The real work starts after we identify an error, which can occur far from where we detect it. When MUST detects an error, it reports the location at which the problem manifests. At this point, a general-purpose debugger is often useful. However, most debuggers poorly support the display of structures that require context knowledge to understand their meaning. For example, debuggers regularly fail to display data structures like C++ STL containers intuitively. The context of debugging MPI applications holds similar challenges as most MPI objects are opaque: the user code only has handles, which are typically integer values or pointers, to represent them. The debugger has no context information about a handle so it displays this unintuitive value. The GNU debugger *gdb* [15] helps to solve this problem by providing an interface to manage the output when printing a variable that can allow access to the context of the variable.

A. Gdb pretty printer

The Python interface of gdb provides a mechanism, to register a pretty printer module. All registered modules are invoked for each variable that gdb prints. If no module prints the variable, the default printer module is used. This mechanism allows context knowledge to determine the display of the variable.

B. Context knowledge

Context knowledge is easily applied to the output for STL containers, as it is in most cases sufficient to filter the output. For the case of MPI handles, such as those for MPI datatypes, the context of the variable is dynamically defined. The context knowledge is bound to the MPI handles by the calls to the datatype constructor functions. MPI-2 defines a mechanism to obtain the context of an MPI datatype by calls to `MPI_Type_get_envelope` and `MPI_Type_get_contents`, which provide access to the parameters of the type constructors. If the datatype is based on other derived datatypes, the returned handle represents a new datatype structure, which must finally be freed. These functions must recursively handle each node of the datatype tree. This approach means repeated memory allocation in application memory space and calls to functions in the application environment. These actions change the application's memory. In general, the debugger should be able to unroll the changes, but we observed that on repeated rollbacks the debugger failed. We need a solution that:

- Ports to all (most) MPI implementations;
- Applies to MPI-1-only implementations like MPICH;
- Has low runtime and memory overhead; and
- Allows the debugger to read data without function calls or writing to application memory.

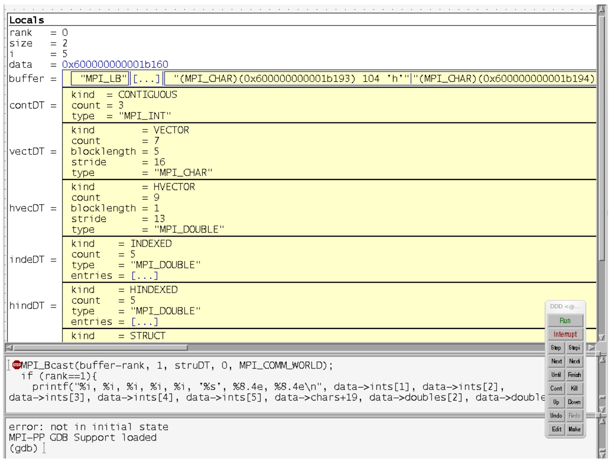
We exploit the MPI profiling interface to achieve these goals. This interface allows interposition of a wrapper library between the application and the MPI library. We save the parameters of each call to an MPI type constructor or destructor into a data structure. This approach achieves our first two goals. The focus of the wrapper functions is therefore on the last two goals. To achieve low memory overhead, we define a structure for each kind of datatype that can record the parameters of the constructor call for this kind of datatype. The parameters are the smallest representation of the datatype without losing knowledge about the datatype – except for compressing the data. The resulting structure corresponds to the meta type-tree. Instances of the same datatype in the tree are covered by one structure in memory. To provide both the wrapper and the debugger pretty printer module fast access to the structure that corresponds to a datatype handle, we register the structures to a hashmap with the datatype handle as key. As the debugger has access to all variables located in the context of the application, the plugin can read the context knowledge for a datatype handle from application memory without changing it.

C. Display information about type handles

Instead of printing the handle value, the debugger can provide useful information about a datatype's shape. We display the parameters of the constructor of the datatype in the form of a C-struct. Thus, any GUI that uses gdb as a backend can profit from the extension without further effort. Figure 5 gives an example for the output of gdb for the datatype introduced in Figure 1 applying our gdb extension. We named the extension mpipp (MPI pretty printer).

```
(gdb) print/r struDT
$1 = 115
(gdb) print struDT
$2 = {
  kind = STRUCT,
  count = 5,
  entries = {{
      blocklength = 1,
      offset = -4,
      type = "MPI_LB"
    }, {
      blocklength = 5,
      offset = 0,
      type = "MPI_INT"
    }, {
      blocklength = 7,
      offset = 47,
      type = "MPI_CHAR"
    }, {
      blocklength = 2,
      offset = 76,
      type = "MPI_DOUBLE"
    }, {
      blocklength = 1,
      offset = 100,
      type = "MPI_UB"
    }}
}
```

Figure 5. Display type from Figure 1 in gdb



```
Locals
rank   = 0
size   = 2
i      = 5
data   = 0x600000000001b160
buffer = "MPI_LB" [...] "(MPI_CHAR)(0x600000000001b193) 104 'h'" "(MPI_CHAR)(0x600000000001b194)
contDT = kind  = CONTIGUOUS
         count = 3
         type  = "MPI_INT"
vectDT = kind        = VECTOR
         count       = 7
         blocklength = 5
         stride      = 16
         type        = "MPI_CHAR"
hvecDT = kind        = HVECTOR
         count       = 9
         blocklength = 1
         stride      = 13
         type        = "MPI_DOUBLE"
indeDT = kind    = INDEXED
         count   = 5
         type    = "MPI_DOUBLE"
         entries = [...]
hindDT = kind    = HINDEXED
         count   = 5
         type    = "MPI_DOUBLE"
         entries = [...]
         kind    = STRUCT
```

```
MPI_Bcast(buffer-rank, 1, struDT, 0, MPI_COMM_WORLD);
  if (rank==1){
    printf("%i, %i, %i, %i, %i, '%s', %8.4e, %8.4e\n", data->ints[1], data->ints[2],
data->ints[3], data->ints[4], data->ints[5], data->chars+19, data->doubles[2], data->double
```

```
error: not in initial state
MPI-PP GDB Support loaded
(gdb)
```

Figure 6. Display datatype and buffer in DDD

D. Display informations about buffers

With the help of context knowledge we can provide a view of the buffer given in MPI communication functions. The Python API of gdb allows one to define additonal commands for the gdb command line. The first approach to display the content of a buffer is a command similar to typecasting in some languages. A call to *mpipp <type handle> <buffer address>* prints the memory content that is described by the datatype that the type handle references. The definition of a new command means that a frontend could implement support for this command to profit from this feature. As we do not want to pass the effort completely to the frontend, we again introduce MPI context knowledge. When a pointer or array is printed and the address is a buffer argument in an MPI communication call near the current position within the application then the buffer is formatted by the corresponding datatype given in this communication call. Figure 6 illustrates the kind of information that can be provided by the extension. It displays the variables known in the local application context (*info locals*). The figure shows an unmodified DDD screenshot with our extension loaded to the gdb backend. The pointer *buffer* is displayed with its content as seen by the current `MPI_Bcast` call.

E. Alternative methods to display the structure of datatypes

As the extension is written in Python, we could provide the information about the datatype in a GUI window that is launched by a newly defined gdb command. The advantage of a GUI window is the possibility of a more natural browsing experience within the data. Our prototype displays the data as a directory tree where the sets from Figure 5 are represented as collapsible nodes and the set elements as children of this node. Also the data of a communication buffer is displayed with the help of the directory tree. The collapsible nodes simplify keeping an overview for large buffer sizes. The large amount of displayable data is a problem for the prototype as we evaluate all information in advance for later display. In a more suitable implementation the information would be evaluated just-in-time, i.e., when the node is expanded.

XI. Future work

A. Connecting must and debugger

We plan to connect the MUST approach to the debugger approach. Instead of generating an error message when an error occurs, we will start an instance of gdb that is connected to the faulty process. The improved gdb enables the developer to investigate the source of the error.

B. Limitations of the extension

Real life applications sometimes use complex type structures. In this case the output quickly becomes too large to review. The possibility to browse down the type tree could become useful. A possible application is the addressing within the datatype as described above. As MUST describes the location of an error, it would be helpful for the gdb extension to understand this addressing as input.

C. Handling other handles

Our gdb extension does not cover other MPI handles. Information about the handles of request, groups, and communicators could be useful for debugging MPI applications.

XII. Conclusion

We presented two components that provide a holistic approach for debugging MPI datatype usage in large scale applications. The first component, integrated into the MPI runtime error detection tool MUST, can detect several common errors when dealing with MPI datatypes. The detected errors include general checks for function arguments, checks for type matching, and checks to detect overlapping communication buffers. We presented scalable algorithms to perform these checks even for complex derived datatypes. Our algorithms use a novel regular expression representation for type signatures and strided blocks for the detection of buffer overlaps. A path representation for correctness errors allows users to pinpoint the results of these automatic correctness checks easily. We demonstrated the scalability of our overlap checks for up to 2048 processes with two widely used benchmark suites, SPEC MPI2007 and the NAS Parallel Benchmarks. During our experiments we detected MPI usage errors in the benchmarks *115.fds4* and *122.tachyon*.

Our second component for holistic MPI datatype debugging extends the GNU debugger *gdb*. This extension provides insight into MPI datatypes when debugging an MPI application. While current debuggers only display the numeric values of MPI datatype handles, our extension can display the datatype's structure. Further, we allow users to print the values that actual communication calls transfer.

Acknowledgments

This article (LLNL-CONF-522011) has been authored in part by Lawrence Livermore National Security, LLC under Contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," http://www.mpi-forum.org/docs/mpi22-report.pdf, April 2009.

[2] "SPEC MPI2007 Benchmark Suite for MPI," http://www.spec.org/mpi2007/.

[3] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A Tool for Model Checking MPI Programs," in *PPOPP*, 2008, pp. 285–286.

[4] B. Krammer, T. Hilbrich, V. Himmler, B. Czink, K. Dichev, and M. S. Müller, "MPI Correctness Checking with Marmot," in *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, HLRS. Springer Publishing Company, Incorporated, 2008, pp. 61–78.

[5] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, ser. Supercomputing '00. Washington, DC, USA: IEEE Computer Society, 2000.

[6] W. Gropp, "Runtime Checking of Datatype Signatures in MPI," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 2000, pp. 160–167.

[7] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 2, pp. 93–100, 2003.

[8] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[9] D. Lecomber, "Debugging the Future with DDT at ORNL," http://www.nccs.gov/wp-content/uploads/2009/06/DDT_ORNL_Tech_Day_1109.pdf, Apr. 2011.

[10] TotalView Technologies, "TotalView," http://www.totalviewtech.com/products/totalview.html, Sep. 2011.

[11] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann, "Flattening on the Fly: Efficient Handling of MPI Derived Datatypes," in *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 1999, pp. 109–116.

[12] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event Based Tools in Parallel Systems," in *IPDPS 2012: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium*, 2012.

[13] M. Lieber, V. Grützun, R. Wolke, M. S. Müller, and W. E. Nagel, "FD4: A Framework for Highly Scalable Load Balancing and Coupling of Multiphase Models," in *AIP Conference Proceedings*, vol. 1281, no. 1. AIP, 2010, pp. 1639–1642.

[14] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," NASA Ames Research Center, RNR-91-002, Aug. 1991.

[15] Free Software Foundation, "Debugging with gdb: The GNU Source-Level Debugger," http://sourceware.org/gdb/current/onlinedocs/gdb/, 2010.